How to Get a Large Natural-Language System into a Personal Computer

Bozena H. Thompson
and
Frederick B. Thompson

Computer Science Department
California Institute of Technology

5215:TR:86

How to get a Large Natural-language System into a Personal Computer

Bozena H. Thompson
and
Frederick B. Thompson

Computer Science Department
California Institute of Technology

5215:TR:86

Presented at the National Computer Conference, Chicago, July, 1985

How to get a large natural-language system into a personal computer

by BOZENA HENISZ THOMPSON and FREDERICK B. THOMPSON
California Institute of Technology
Pasadena, California

ABSTRACT

The answer to the question of how to get a large natural-language system into a personal computer lies in the paging architecture of the system. The key is to use the input sentence, in conjunction with the lexicon and grammar table, to identify the minimal segments of both object code and data that must be brought into main memory. Once such a maximally paged architecture has been effectively implemented, it has wide ranging implications for process integration, networking and knowledge base distribution, and for the software engineering environment. The Natural Access System optimizes this architecture and exploits these implications.

The Natural Access System is a large natural-language system. It is now implemented and running on a personal computer (PC). This paper tells how we were able to get such a large system on such a little computer.

THE OBJECTIVES OF NATURAL ACCESS

The Natural Access System has evolved over a number of years with the object of providing truly natural access to the computer for intelligent people who may not be programmers or even computer literate. Because of the academic setting of our research, we have been able to take a fresh look at this problem, and to guide our research through considerable experimentation.[1-3] We have assumed that most people will have direct access to computers in civil and business organizations, in research labs, on engineering floors, among management staffs, and in their homes. The forms and facilities of this access are still evolving in response to the question, what is the proper form for this access and what are the requirements for a computer software system that will provide natural access to computers?

Experience with existing systems makes it abundantly clear that almost every application of computers is a special application not adequately served by any single general purpose system, and that the requirements of any specific application are constantly in flux, thereby necessitating constant updating and extending of any system implemented for its support. Yet, it is not economically viable to program a new system from scratch for each application and to constantly retrofit that detailed special purpose design. What is more tenable is a system design that serves a defined range of applications, and within this range, is readily adaptable to each specific application; a system which in its design supports its own customizing and recustomizing to specific requirements. Today, the computer software industry is structured to respond to these realities as evidenced by the proliferation of software firms that specialize in narrow applications areas, and others in the development of sophisticated tools applicable to a variety of uses. A general system must accommodate itself to this reality. It must also attend to the problem of software engineering. The design of the Natural Access System has been undertaken in light of these considerations.[4,5]

The domain of applications for which the Natural Access System is designed can be roughly described as those where

1. the individual's interaction with the personal computer dominates the computing task;
2. the tasks are of a reasonable degree of complexity requiring more than simple record keeping; and
3. an individual's interaction with the computer impinges on other people. Thus the computer acts as a medium for communication.

This domain specifically excludes *compute-bound* applications such as those arising in the physical sciences, and applications such as airline reservations systems where the capability to handle large volumes of simple transactions is required. Applications typical of the domain we intend to cover are local area networks serving business organizations, computer support that provides intelligent interfaces to the experimental apparatus of research teams, the media for computer-aided design and the coordination of design in an engineering shop, and the household computer with links to banks, stores and wide-ranging sources of information.

In order for a computer system to be responsive to the needs of an organization, it must contain a great deal of "knowledge" about the domains with which the organization is concerned. Knowledge can take the form of databases whereby the structural linkages establish relationships among database entities, procedures that express implicit meanings not discernible in structure alone, and assertions wherein implications are developed through symbolic manipulation. A "knowledgeable" computer must be able to integrate these forms with the understanding of succinct instructions; an understanding that prompts it to marshal data, processes and assertions in complex ways that respond to the immediate needs of the user.

In many organizational environments, the knowledge base is rapidly changing. Its maintenance becomes one of the major activities of the organization itself. This is true of corporate information systems, systems for the coordination of public agencies, and the many systems that support management, engineering and military staffs. Maintenance of the knowledge base is not a single level of activity since managers, clerks, engineers and applications programmers will all be actively interacting with the knowledge base. The knowledge base will indeed be their integrated, dynamic group memory; the information context that gives substance and meaning to their otherwise diverse activities. The analyses of the dynamics of these varied interactions must be a ubiquitous aspect of the design of such a system.

In a complex organization of professionals such as the corporation, research laboratory, or military staff, there can be no single knowledge base, but many, each with its own rate of change, content and responsible agents. On the other hand, these various knowledge bases are by no means independent. Figure 1 illustrates the relationships between knowledge bases that may exist in a simple organizational structure.

In a large organization such as a multi-national company,

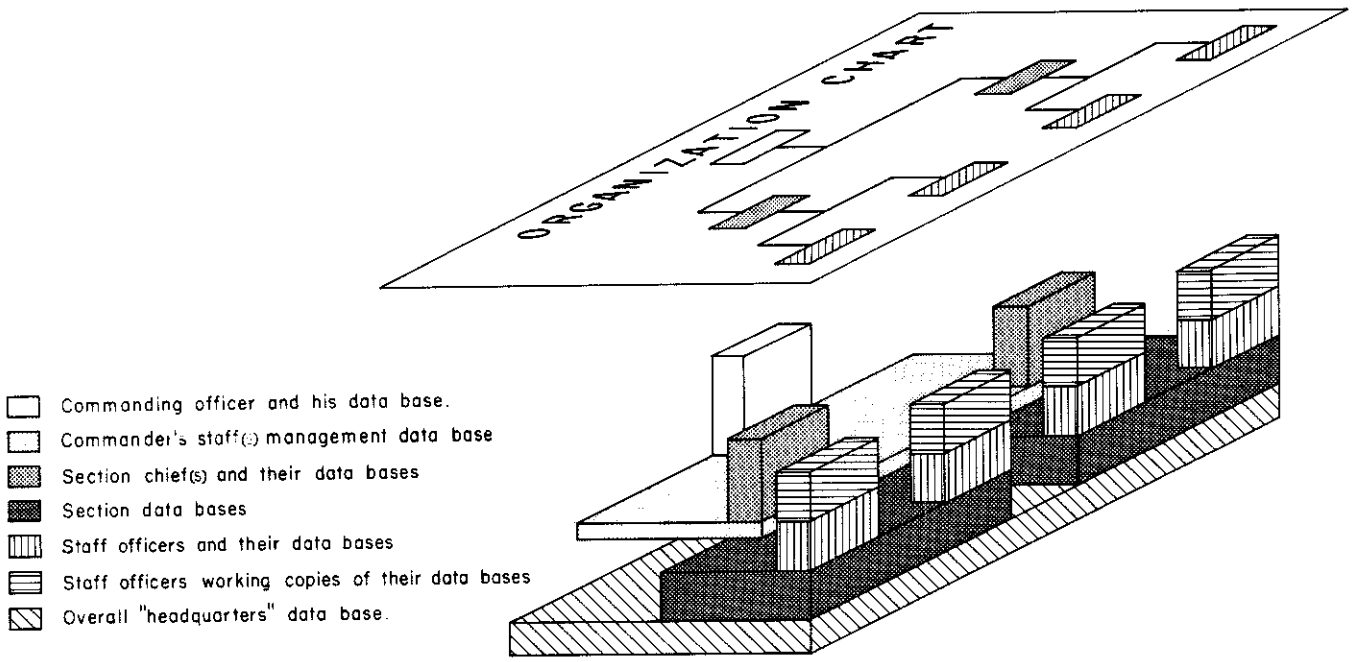

Figure 1—Relationships between contexts in a small organization

the intensity of inter-knowledge base activities also has spatial dynamics, because there is less communication between distant corporate entities. The nature of interactions between offices that are vertically related (i.e., one being organizationally subordinate to the other) is different than that between two laterally related sister offices. Therefore, a system for such organizations must be able to support many knowledge bases and incorporate means for their interaction.

What should the form of communication be between a member of the organization and the system? There are many forms for human–computer interface under development. However, the context of the professional's interaction will largely determine the appropriate form; designing a piece of machinery or developing the load plan for a ship clearly calls for graphic interaction; standard queries of a repetitive nature call for the use of tableaux or forms; when the computer is required to respond through a complex decision structure to relatively few but varied inputs, a system-guided dialogue is desirable; in a word processing environment, simple single-key instructions with cursor control are called for. All of these appropriate interfaces should be available for users to invoke as a natural aspect of their own structuring and development of various knowledge bases. Further, the user should be able to employ various forms of interface to the same underlying knowledge base, fully integrating graphics, images, text and dialogue within that knowledge base as deemed appropriate, when all concern the same domain of data and process.

These two requirements, the ability of professionals to arrange their own forms of interface, and the integration of available interfaces with each other and with the knowledge base, suggest a single form of interface, prior to the invocation of the others, which is sufficiently flexible, expressive, and natural to be used both as an interface form in itself, and in the design and specification of other interface forms. In a system that gives truly natural access, this prior interface should be a simple dialect of English. English then becomes the primary language by which the user can customize his* forms of interaction, and to which he can fall back when the more succinct forms he has evolved are no longer sufficient in light of changing needs.

THE NATURE OF CONTEXTS

We will give more specific structure to this general notion of knowledge base, referring to it as a *context.* Roughly speaking, a context encompasses a vocabulary, language, definitions, database and possibly special procedures and extensions. A professional at the console will usually be interacting with the computer in such a context. Each professional will probably have a number of contexts; perhaps one encompassing administrative matters for keeping work logs, personal budgets and schedules, files of memos concerning assignments, and progress reports; one containing the specifications, design ideas and requirements for each of the projects in which he or she is participating; and possibly a bibliography. At any given moment, one may have made a copy of a working context, and be trying out some new ideas using the copy so that the original is not irretrievably destroyed.

The idea of a context can more easily be grasped by considering how such a context can be created in a Natural Access System. Initially, the system contains just one context, called the *BASE Context.* BASE contains a simple dialect of English

* The masculine pronominal forms are used in this paper simply because of the brevity of *he.*

as its primary language, a math and statistical package, text, image and graphic packages, means to add vocabulary and definitions for adding and changing data, and other ingredients of a fully-implemented knowledge-based working environment. The vocabulary of the BASE context contains all of the "small" words (i.e., *and, what, was, exceeds,* etc.), terms such as *square root* and *average,* and commands such as *create, list,* and *display*. To create a new context, the professional enters the command:

)Base . . . on ---

Suppose his organization, called *Corporate,* has a general context containing addresses and phone numbers, cost-center rosters and task assignments, and further suppose that he wishes to have this data at his fingertips, and to add to it his own administrative records for his private use. He may then type

)Base my admin on Corporate
)Enter my admin

and proceed to use this new context for his own purposes. Referring back to Figure 1, when one context rests on another, this basing relationship is indicated. If Context B is based upon Context A, any changes in A will automatically and immediately be reflected in B; however, changes in B will not affect A in any way. One context can also be based upon many contexts, thereby allowing it access to all of their knowledge bases. Many contexts may also be based on a single given context.

THE LANGUAGE PROCESSOR OF THE NATURAL ACCESS SYSTEM

A central aspect of the concept of a natural access system is that of the naturalness and flexibility of the user language. If the user is indeed able to use his natural language, the language processor for that language is never far from system architectural concerns. We first examine the nature of that processor.

The language processor of the Natural Access System is a simple syntax directed interpreter. The system is based on *compositional semantics* and it has *procedural semantics.* Semantics is that part of the specification of a language that deals with meaning. Compositional semantics is a way to interrelate the meaning of a sentence to the syntactic structure of the sentence. Compositional semantics assumes that the syntax of the language is given in the form of general rewrite rules, each associated with a semantic interpretation procedure. Applying the rules to the syntactic analysis of the sentence, a procedure called *parsing,* results in a parse graph. The semantic procedures associated with the applicable grammar rules are composed or *compiled* using the parse graph, and the result evaluated. Careful exposition of this concept of compositional semantics shows that the role of part of speech in rules of syntax is to guarantee that the arguments of the associated semantic procedures are of the correct type (in the sense of "type" in such programming languages as Pascal). For example, consider the following rules (where "num" stands for "number", "att" for "attribute" and "comp" for "comparator")

R1: ⟨num⟩⇒⟨num att noun⟩ "of" ⟨class noun⟩
R2: ⟨num⟩⇒⟨article⟩ " " ⟨num⟩
R3: ⟨num⟩⇒⟨num⟩ " + " ⟨num⟩
R4: ⟨clause⟩⇒⟨copula⟩ " " ⟨num⟩ " " ⟨comp⟩ " " ⟨num⟩
R5: ⟨sentence⟩⇒⟨clause⟩ "?"

and the sentence

Is 40 greater than the number of students + 5?

We assume that a preprocessor, referring to the lexicon, has parsed this sentence as follows:

⟨copula⟩ ⟨num⟩ ⟨comp⟩ . . . . . ⟨article⟩ ⟨num att noun⟩
Is 40 greater than the number
⟨class noun⟩ ⟨num⟩
of students + 5 ?

The above rules result in a *parse tree* for this sentence, shown in Figure 2.

In compositional semantics, a semantic procedure is associated with each rule of grammar. For example, consider the rule

R3: ⟨num⟩⇒⟨num⟩ " + " ⟨num⟩

with the associated semantic procedure PLUS PROC. Clearly, PLUS PROC adds the two constituent numbers. By this grammar rule, we have the parsing

⟨num⟩ . . . . . .
⟨num⟩ ⟨num⟩
3 + 5

and thus the meaning of "3 + 5" is

PLUS_PROC(3, 5) = 8.

Now associate with each of the above five rules the following procedures.

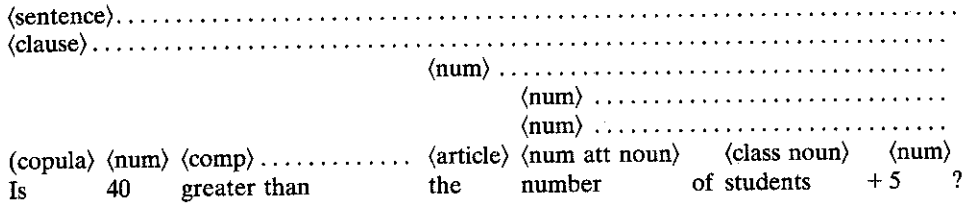

Figure 2—Parse tree for the sentence

R1: ⟨num⟩⇒⟨num att noun⟩ "of" ⟨class noun⟩ : FIND_ATT
R2: ⟨num⟩⇒⟨article⟩ " " ⟨num⟩ : NO_OP
R3: ⟨num⟩⇒⟨num⟩ " + " ⟨num⟩ : PLUS_PROC
R4: ⟨clause⟩⇒⟨copula⟩ " " ⟨num⟩ " " ⟨comp⟩ " " ⟨num⟩ : COMP_PROC
R5: ⟨sentence⟩⇒⟨clause⟩ "?" : OUT_PROC

It is not surprising to find that the semantic procedure associated with some rules is the NO OP procedure; such rules govern *function* words that play a purely syntactical role (*the* in some linguistic contexts may not be a function word). Referring to the above parse tree, we see that the meaning of the example sentence is given by the functional composition

OUT_PROC(COMP_PROC([40], PLUS PROC (FIND_ATT([number], [students])), [5])))

Note that we have associated the FIND_ATT procedure with the rule:

R1: ⟨num⟩⇒⟨num att noun⟩ "of" ⟨class noun⟩ : FIND_ATT

The meaning of the word *number* is given by means of the procedure that counts a class and returns the number of elements in the class. A pointer to this procedure is found in the lexicon as part of the definition of *number*.

Rule R1 would also apply to the phrase

⟨num⟩		
⟨num att noun⟩		⟨class noun⟩
⟨unary operator⟩	⟨num att noun⟩	⟨class noun⟩
average	age	of students

In this case, the first constituent to which R1 applies is not a simple procedure as in the above example, but a more complex structure whereby *age* is most likely given by a table; FIND_ATT uses this table to find the set of numbers which are the ages of students. FIND_ATT then examines the unary operator *average,* and finding it to be given by a procedure, applies this procedure to this set of numbers.

Thus we see that the meaning of a word or phrase can be given by a pointer into the database by means of a procedure, or even a complex structure of database entries and procedures. For this reason, *compositional semantics,* when encompassing this possibility, is also referred to as *procedural semantics.* In discussing programming languages such as Simula, where the meaning of semantic entities can have a mixture of structural and procedural aspects, the term *object-oriented language* has been used.

The language processor of the Natural Access System is a general rewrite rule–procedural semantics processor. It is general in that the syntax of any well defined language can be specified by a general rewrite rule grammar. On the other hand, procedural semantics is not sufficient to define the semantics of useful dialects of natural language because the understanding of a natural language utterance depends on the context of the dialogue in which the utterance occurs. Methods for incorporating this dependence on dialogue context into a natural access system constitute one of the most important and active areas of artificial intelligence research. The concepts of Frames (introduced by Minsky),[6] Scripts and Mops (introduced by Schank),[7] and Partitioned Networks (introduced by Hendrix)[8] are significant contributions to this area of research. The Natural Access System falls short of incorporating methods as far reaching as these three. However, in the handling of anaphora and case frames of verbs, it goes well beyond procedural semantics.[9] It does so by explicitly handling these aspects in procedures that are called in the language processing module but independently and in parallel to the central parsing and semantic processing procedures.

The Natural Access System (NAS) handles many languages, both dialects of natural languages such as NAS English and NAS French,[10] and programming languages used in the NAS Metalanguage environment. One of the central architectural features of the Natural Access System is that all of these cases use the same language processor. Therefore, all NAS languages are syntax rule-based and procedural, and may be object-oriented in the above sense. They differ only in their syntax rules and associated semantic processing procedures. Thus a language is implemented in the Natural Access System by declaring its syntax (any general rewrite rule grammar), and for each syntax rule, the associated semantic procedure. This has great advantages as described below.

The Natural Access System is sentence-driven. One can think of the Natural Access System as operating in the following paradigm.

1. The system types a "⟩", indicating that it is ready for user input, and waits for the user to respond.
2. The user enters a sentence, or more generally any string ending in an *end of entry* key. (We will often refer to such a string as a *sentence* even when it does not parse to ⟨sentence⟩.)
3. The system responds by processing the sentence, displaying the results on the user's terminal, and cycling back to the first step.

Many aspects of Natural Access System processing are organized around this paradigm. Here are some typical examples.

1. During the processing of a sentence, the processing procedures may make use of various forms of temporary work space; at the end of sentence processing, these are returned to the general work space pools.
2. If the user input is not a true *sentence,* the system tries a variety of corrective procedures (i.e., spelling correction), assuming that the input by itself was supposed to constitute a meaningful expression.
3. The user might inadvertently enter a string which, for one reason or another, takes an inordinate amount of time to process. In this case, the user can at any time press the BREAK key and abort the current processing. In this circumstance, the system "cleans up" its working environment, readjusts itself, and is ready to receive the next sentence.

CONSIDERATIONS IN THE DESIGN OF THE CENTRAL ARCHITECTURE

Central to the design architecture of the Natural Access System, as with any "next generation" system, is the difficulty such a system would have were it to function within one hardware level of memory. The discrepancy between *main memory* and *peripheral memory* will continue to be significant, and must be a consideration in the management of the object code and knowledge base.

Because the total body of just the object code involved in a system such as the Natural Access System is too large to fit in main memory at one time, it is tempting to nibble away at integration through the expedient of increasing main memory. But dependence on this stopgap method impedes progress toward a rational solution of the problem. The object code of a truly integrated system includes not only the operating and language processing subsystems, but the semantic procedures of the primary languages, text processor, image processor, database management system, display management procedures, user language-specific semantic procedures, and the many application-specific procedures developed by programmers for their users' domains. The integrity of the user context depends on the unlimited capacity of the code body to accommodate extensions as the user community matures and the size and complexity of the computer's tasks grow. Segmentation of this code thus becomes a major architectural decision.

The solution to this problem for current generation systems is epitomized by the UNIX system. The resident UNIX code is small. A UNIX system environment has a number of major code bodies (*shells*) which can each be loaded into main memory and given control and which communicate through a common filing system. These shells are typically large enough to embody a significant form of user interface.

Typical shells such as word processors, database management systems, or application programs that handle prescribed yet versatile domains of user considerations can be characterized as *process oriented.* If the user is a system programmer, process orientation fits the conceptual organization of his task well. He simply writes and edits code, compiles it, and applies it to a specific domain. His interests are often with new systems having small programming infrastructures and programming languages that are quite free of domain-specific utilities. There is no need in his world to mix media; source code is in a fixed medium, namely text (though it may concern the manipulation of several media).

Indeed, the UNIX architecture serves computer research and development laboratories well. However, segmentation based on process orientation does not fit the conceptual organization of end users. Text, pictures, data, display formats, statistical processing, and distributed sources of data commingle in the user's conceptual world. Thus in the computer system context with which the user is engaged, such requests as

)Display a bar graph of the variation in net sales of each sales region over the last three months.

and

)How many originators of memos referring to ICOT were there in each organization?

require procedures from the display management package, statistical package, telecommunications package, database management package, and text processor, albeit from only a small fraction of the procedures in any one of these packages. For the user to exit from his word processor, enter his electronic mail system to retrieve the net sales of his various sales regions, invoke his statistical shell, and finally go to his display formatter to see the desired bar chart is not natural access. To hide these transitions from the user in yet another shell, but one that leaves him little flexibility and does not relieve the situation, is not a good solution, either.

Segmentation based on processing function is also not germane to the needs of the applications programmer. He does not program a new system from scratch, but extends the existing system capabilities for his user clientele. The application area he is building or extending, depending on what part of the application programming chain he is in, already has an extensive infrastructure of database processing and special display management utilities embodying a great deal of domain-specific knowledge. Much of his programming consists of calls to these utilities. In present practice, he has no choice of programming language. In a UNIX environment, he is extending an existing application shell, but unless he is willing to give up control of paging faults and use a virtual memory (giving up good response times by avoiding thrashing even though his knowledge of his particular application would allow him to do so anyway), he is limited in the extent of the application package he can put together by the capacity of main memory. Even if he chooses to use virtual memory, a choice often forced upon him by the available compiler for the language, and to pay the unknown price for thrashing when his application gets realistically large, he is faced with yet another dilemma; for when he wishes to use small parts of procedure packages already existing in other shells, he must either reprogram them, or use utilities that swap shells for him, making available both the useful procedure and its brothers, sisters, aunts and uncles who are "swapped along" for the ride.

In consideration of the context of the end user and that of the applicatons programmer, the Natural Access System uses a distinctly different segmentation architecture as presented next.

CENTRAL ARCHITECTURE OF THE NATURAL ACCESS SYSTEM

When a user of the Natural Access System enters a sentence, the entry is parsed. As a result of this parse, the relevant semantic procedures associated with the rules, and perhaps procedures associated with words in the sentence, are identified. Calls to these, and the utilities they may call, are compiled and executed in the processing of the sentence. With the exception of the small resident code, no other procedures are

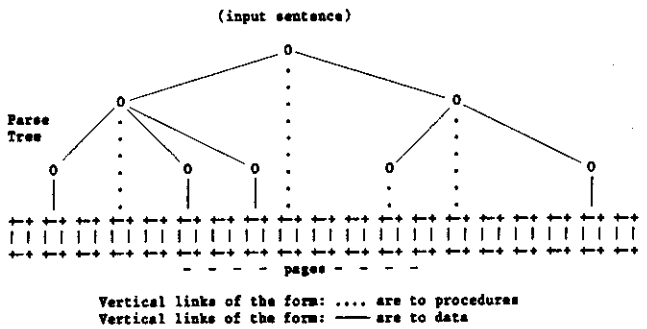

Figure 3—Relationships between parsing and page loading

relevant, and no other procedures need be in main memory while the sentence is being processed (indeed, these may not all need to be in main memory at the same time). Access will presumably be made to database entities. Thus, the relevant database entities must also be brought into main memory, but these too constitute a very small part of the entire database.

In the Natural Access System, procedures and data entities are paged individually on the same peripheral memory pages, and into the same page-frame area in main memory. To this end, the Natural Access System is its own page manager. It can delegate control of the locking and marking of pages to both the language processor and the individual semantic procedures and utilities. For example, the grammar table is made up of a binary search tree and a rule definition part, all of course on pages. When the parser is called, it can load and lock the entire binary tree while individual syntax time procedures are brought in as needed in conjunction with selected rules. This drastically reduces parsing time otherwise required if parts of that tree were swapped with all syntax procedures as a package every time one of them were needed. Procedures can determine how many page frames are dynamically available to them and optimize their page loading and locking at the last moment by making use of information available about the size and disposition of the structures in their calling sequence as well as the processes they will be carrying out. Such information is not available to a system-level segment manager.

Figure 3 gives a schematic view of the relationship between parsing and page loading. In processing the input sentence, only the pages linked directly or indirectly to nodes in the parse tree need be brought into main memory. The number of pages available on peripheral memory is very large. Several hundred of those pages will contain semantic procedures; many others will contain specialized utilities such as database, graphic, screen management, text, and image processing utilities. Realistically, database data pages will run into the thousands. Knowledge bases will use pages containing complex structures; most of the page leaves will be page pointers to other structures. A given sentence will need to call upon only a small number of these.

Secondary design decisions that are crucial to the viability of this architecture involve

1. the paging algorithms;
2. the section of what system code is to be resident and what is to be paged;
3. the size of the page frame area of main memory; and
4. knowledgeable optimizing algorithms that are built into procedures for controlling page loading.

The test of this architecture is response time. Actual tests of the system indicate that the processing of a typical English query requires something in the order of 200 page loads. Average throughput response time on a Motorola 68010 chip-based computer for sentences such as

"List the destination and cargo-type of each shipment on the Tokyo Maru."

is four seconds; for the sentence

"What is the average population of cities?"

where there are perhaps 1500 cities, it is 20 seconds. These times reflect the following layout of main memory as required by the Natural Access System

1. resident NAS code	150K bytes
2. page frame area @ 1K	120K bytes
3. list processing area	100K bytes
4. miscellaneous work areas	30K bytes

The advantages of the NAS architecture is evident. All of the capabilities provided by the system are equally available. These capabilities can be extended in any direction by simply incrementally adding new syntax rules and their associated semantic procedures. The user's input sentence is all that is needed to orchestrate the loading of the data and procedures necessary for the processing of that sentence. The repertoire of capabilities available for that processing is not limited by the size of main memory, but rather by the effectively unlimited size of peripheral memory. Thus, the notion of integration, as so often discussed in personal computer literature, is achieved to its fullest extent.

NETWORKING

Once all procedures and data are paged on the same pages, and paging is managed by the Natural Access System, another opportunity becomes available. The page is an ideal packet for communication, and the page address is ideal for managing that communication. A Natural Access System installation can be viewed as a network of NAS stations, each consisting of a computer (it is appropriate to think of these as personal computers) and at least one hard disk with a minimum capacity of, say, 20 megabytes. We are presently using a page size of 1024 bytes. Thus, such a disk would hold 20,000 pages. We can also think of stations as grouped into clusters, each with a designated agent, and agents tied together in a wider net. (The cluster grouping allows stations to be added to or taken out of a cluster without reconfiguring the whole net.)

In the Natural Access System, a page pointer consists of a

1. cluster number (1 byte);
2. station number within a cluster (1 byte);
3. volume number relative to the station (1 byte);
4. block number on the volume (3 bytes); and
5. page offset on the page (2 bytes).

Thus, the bound on the amount of network information accessible to any station is the prodigious figure of 2^{48} pages or over 10^{17} bytes. Page pointers are ubiquitous throughout the data and system structures (e.g., grammar table, and database) of the Natural Access System. However, only the paging module "knows" anything about the structure of these pointers. When a system or semantic procedure requests a page, the paging module at the station checks for the page in main memory. If the page is not found, it checks for it on the station's own volumes. It may need to go to another station or, through its agent, to another cluster for the page. The user will never know, except perhaps for an unaccustomed delay in response, where the page is coming from. How such page addresses are distributed will be clarified below. The paging module, on the other hand, is unaware of whether the page it is retrieving contains object code, data, text, digitized voice, pixels, or perhaps some specialized data structures used in the concerns of a small community spread around the world.

THE INTERACTION BETWEEN PAGES AND CONTEXTS

In discussing the objectives of natural access, we introduced the notion of an NAS context. Such contexts were introduced as encompassing vocabulary, language, definitions, database, and possibly special procedures and extensions. New contexts are created by invoking the basing operator. For example, let us say that the Context BB is based on the Context AA.

)Base BB on AA.

To understand how Contexts and pages interact, recall that the grammar table of AA is made of two parts, the binary search tree and the rule definitions, both on pages. Language extensions such as user-introduced vocabulary and applications programmer-introduced grammar rules are added to this table. As long as context BB is based on AA, new entries in the binary search part of BB's grammar table, which are copies of those in the corresponding binary search part of AA's grammar table, are introduced. The pointers in AA's binary search tree point into its rule definitions which in turn point into AA's knowledge base. On the other hand, in BB's copy of AA's binary search tree, the pointers point into AA's rule definitions. As a result, the binary search part of BB initially appears exactly like that of AA to the end user. Of course, as extensions and changes are made to BB by the user, BB's own rule definition section begins to take shape, and BB evolves into a distinct context with its own knowledge base, but always remains based on AA.

The grammar tables and knowledge base that lies under them are on pages. Thus, from the point of view of the paging module, the addressability of the knowledge base of AA from BB is insured by the fact that the page addresses of AA's rule definitions are copied into the binary search tree of BB by the basing operation.

To illustrate this point, imagine that the Sales Division of a corporation is in charge of the corporation's Eastern and Western Sales Regions. Let Central Sales be the working context of the Sales Division, containing its local records, etc. Suppose the Sales Division operator types

)Base Central Sales on Eastern Sales
)Base Central Sales on Western Sales

Now, Central Sales contains pointers to pages that reside in the offices of the Eastern Sales Region and the Western Sales Region. Their contexts are kept up-to-date by their respective staffs so that when the manager of Sales Division asks the system to

)Display a pie chart of the sales of each product line for each sales region.

the paging module in the Sales Division computer "knows" it must retrieve the necessary pages of the respective offices of the two sales regions. It also knows the file and page numbers of these pages, because these file and page numbers were passed to Central Sales at the time of basing. Roughly speaking, this is how addressability is handled for the *distributed* database of the sales organization of the given corporation. For a more complete discussion, see the work of Yu.[11]

THE SOFTWARE ENGINEERING ENVIRONMENT OF THE NATURAL ACCESS SYSTEM

To extend a user's context with a new capability, the applications programmer adds a new syntax rule and its associated semantic procedure. For example, suppose an applications programmer wishes to add a new numeric function which his user, a banker, will refer to as *projected average return.* The syntax rule to be added is

RULE proj_avg_ret
⟨num att noun⟩⇒"projected average return"
proj_avg_ret_proc

where proj_avg_ret_proc is the associated semantic procedure which the applications programmer then writes. Once this rule and procedure are added to the user's context, the user can ask the system to

)Display a bar graph of the projected average return of each of the high tech stocks of Mr. R. C. March.

Associated with each user context is a corresponding applications programmer's context which we call the *meta-context.* Figure 4 illustrates these relations in the software engineering environment.

When a context such as AA is based upon another, say BB, second contexts are created, namely meta-AA, based on meta-BB. Just as all user contexts are ultimately based upon the BASE context which contains NAS English, all meta-contexts are ultimately based on META-BASE, which contains Pascal. When the applications programmer wishes to

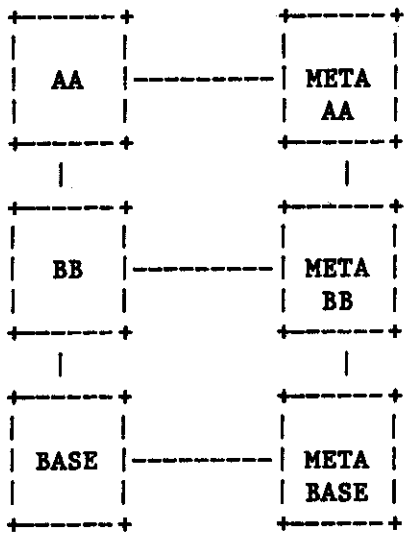

Figure 4—Contexts and their associated meta-contexts

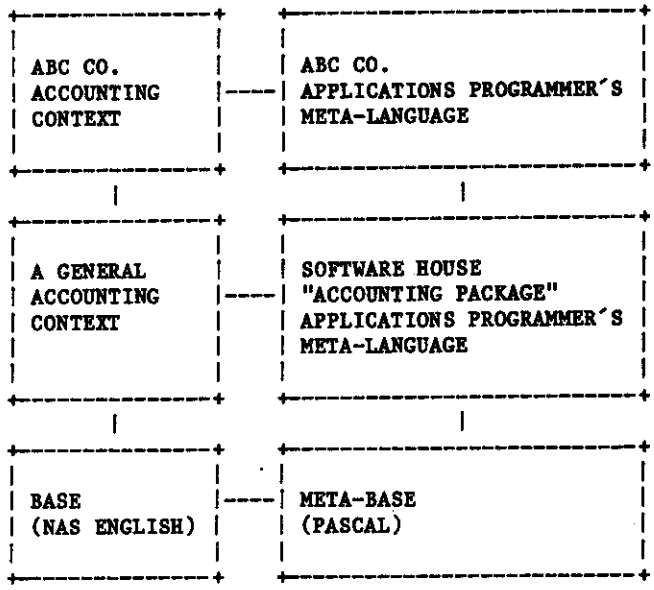

Figure 5—Steps in the applications programming chain

add a rule to Context-AA, he enters the associated meta-context, META-AA. Typing "RULE," and the name he will use in referring to this rule, he is prompted for the syntax and semantics.

```
)RULE proj_avg_ret
)Syntax: ⟨num att noun⟩⇒"projected average return"
)Semantics: procedure proj_avg_ret;
            begin
            ...
            end.
```

The meta-context then

1. collects the semantic procedure;
2. adds the necessary INCLUDEs for system types and variables, utilities, etc.;
3. calls the Pascal compiler;
4. puts the resulting compiler-produced text as a text object in the meta-context's lexicon;
5. takes the object code and links it into the NAS resident code;
6. puts the linked code onto pages; and
7. puts the syntax into the grammar table of the user's context with the page address of the semantic procedure as "semantics."

Of course, META-BASE has a much more extensive software engineering environment.[12] It is driven by the syntax and semantic procedures of the "metalanguage," and is processed like any other NAS language by the language processor (thereby sharing such services as the spelling corrector, definitional utilities, etc.). As one might imagine, the metalanguage includes a wide range of debugging tools, and the syntax and semantics for rule addition illustrated above.

The rule-adding capability allows the applications programmer to extend his user's language. He can also extend his own language with PROCs and MACROs, thereby providing highly tailored programming environments for special application domains. In this way, application programmers who are low in the applications programming chain, as perhaps in the applications areas of computer companies, can provide wide-ranging and powerful utilities with specialized domain knowledge which can be used via basing by those on the applications programming staffs of customer companies (see Figure 5).

It is this metalanguage and its many extensions that are the interactive domain of the applications programmer. This brief introduction is intended to simply give the flavor for this environment; the experienced programmer can project many of the features that are there.

THE ROLE OF SOFTWARE DEVELOPMENT COMPANIES

Beyond the end user community and the applications programmer community, there is a third community concerned with application systems that must be taken into consideration, namely the software development companies. This community, large and rapidly growing, constitutes an important organizational mechanism for the free and competitive use of our intellectual and management resources in the information age. Under a UNIX-type system, the products of this community take the form of additional shells marketed in the form of disks and cassettes.

Corresponding products in the Natural Access environment presently consist of two main programs, one that boots the system in a new installation, and one that proceeds to run it. The main functions of booting are to format the paging data set and initialize the BASE-context grammar table with NAS English and the metalanguage. To this end, both languages are transported as syntax files, and as separate files containing

object code modules for the respective resident code, non-resident utilities, and semantic procedures. System utilities at boot-time build the syntax into the grammar table, page the non-resident code, and link the two together. It is these same utilities that are invoked by the metalanguage after it has called the compiler on an applications programmer's introduction of a new rule.

It is now clear that the software development house will market its package in the form of a disk containing the syntax and object code for the semantics of the new capability it is introducing. The same utilities that are used at boot time are then called to extend a given context with these capabilities. The user, or his applications programmer, enters the meta-context associated with the context he wishes to extend, types

)Extend with ⟨file or volume name containing the application package⟩

and the new capability, completely integrated with the previously existing capabilities, vocabulary, and database of the user's context is now available. This new capability will then be passed on to new contexts that are subsequently based upon the extended context. Of course, the software company would not wish to reveal the metalanguage it has developed for producing this and possibly related packages for this application domain, nor the source code for the package's semantic procedures.

The packages that will be marketed will have a far wider range than those we see in retail computer outlets today. The Natural Access System already includes text, graphic, and image processing, all as an integrated part of NAS English, and on pages thus amenable to transmission within a NAS network. Couple this with the digitizing camera (soon expected to be available from both Canon and Sony), the processing of digitized speech, and the predicted breakthrough in both the speed and cost of digital communication,[13] and the applications possibilities proliferate beyond any bounds.

In considering the preparation of packages, one should remember that all of these processing capabilities (English, text, and image processing) can be assumed to already be in the user's context. For example, a retail department store, using the *Catalogue Preparation Metalanguage Package* it has purchased from a software company, and a digitizing camera, can quickly prepare a catalogue package for its next sale. The syntax and semantics for ordering will reflect specialized knowledge of the catalogue. The telephone number for automatic dialing is also built in. The store then distributes the catalogue to each of its charge customers in the form of a disk that includes the customer's account information and address. The customer can then type

)base catalog on BASE
)extend catalog from disk drive

and then use word processing commands to find desired items, including their pictures, and order the product(s) by typing

) Send me a silk blouse, item number 3B64, size 14.
) What color (white :::, blue ///, or beige %%%): blue
One Anne Klein II silk sports blouse, size 14, color blue
will be sent out today to:
Mrs. John J. Jones
777 Oak Street
My Town, Maine
$94.95 plus $5.10 tax
will be added to your account.
Thank you for your order.

CONCLUSION

The answer to the question of how to get a large natural-language system into a personal computer lies in the paging architecture of the system. The key is to use the input sentence, in conjunction with the lexicon and grammar table, to identify the minimal segments of both object code and data that must be brought into main memory. Once such a maximally-paged architecture has been effectively implemented, it has wide-ranging implications for process integration, networking, knowledge base distribution, and the software engineering environment. The Natural Access System optimizes this architecture and exploits these implications.

REFERENCES

1. Thompson, B. H., and F. B. Thompson. "Introducing ASK: A Simple Knowledgeable System." *Proceedings of the Conference on Applied Natural Language Processing, ACL and NRL.* Santa Monica, California 1983, pp. 17–24.
2. Thompson, B. H., and F.B. Thompson. "ASK Is Transportable in Half a Dozen Ways." *ACM Transactions on Office Information Systems,* April 1985.
3. Thompson, B. H., and F.B. Thompson. "Shifting to a Higher Gear in a Natural Language System." *AFIPS, Proceedings of the National Computer Conference* (Vol. 50), 1981.
4. Thompson, B. H., and F. B. Thompson. "Customizing One's Own Interface Using English as Primary Language." *Proceedings of the South East Asia Regional Computer Conference.* Hong Kong: Hong Kong Computer Society, 1984, pp. 15.1–15.16.
5. Thompson, B. H., F. B. Thompson, and T. P. Ho. "Knowledgeable Contexts for User Interaction." *AFIPS, Proceedings of the National Computer Conference* (Vol. 52), 1983.
6. Minsky, M. "A Framework for Representing Knowledge," in P. Winston (ed.), *The Psychology of Computer Vision.* New York: McGraw-Hill, 1975.
7. Schank, R. C., and R. P. Abelson. *Scripts, Plans, Goals and Understanding.* Hillsdale, N.J.: Lawrence Earlbaum, 1977.
8. Hendrix, G. C. "Encoding Knowledge in Partioned Networks." In N. Findler (ed.), *Associative Networks.* New York: Academic Press, 1979.
9. Trawick, D. J. "Robust Sentence Analysis and Habitability." Ph.D. dissertation, California Institute of Technology, 1983.
10. Sanouillet, R. "ASK French, a French Natural Language Syntax." Master's thesis, California Institute of Technology, 1984.
11. Yu, K. I. "Communicative Databases." Ph.D. dissertation, California Institute of Technology, 1981.
12. Thompson, B. H., and F.B. Thompson. *Natural Access to a Personal Computer,* forthcoming.
13. Sussenguth, E. H. "MIPS & BPS—Communication Abundance for All?" *Proceedings of the South East Asia Regional Computer Conference.* Hong Kong: Hong Kong Computer Society, 1984, pp. 9.1–9.16.